1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery`:

```
algoritmo mystery(A: Int[1..n]) --> Int
  bh = new BinaryHeap()
  k = 0
  i = 1
  while (i <= n)
    bh.insert(A[i])
    k = k+1
    i = i*2
  endwhile
  for j = 1 to (k-1)
    bh.deleteMax()
  endfor
  return bh.findMax()
```

Soluzione L'algoritmo `mystery` esegue operazioni a costo costante a meno delle operazioni `insert` e `deleteMax` su binary heap, che hanno entrambe un costo logaritmico rispetto alla dimensione dell'heap. Il corpo del `while` viene eseguito $\lfloor \log n \rfloor + 1$ volte in quanto l'indice i inizialmente ha valore 1 e viene raddoppiato ad ogni ciclo fino a che non supera n. Ad ogni ciclo viene aggiunto un elemento nell'heap. Essendo il costo di ogni `insert` logaritmico rispetto alla dimensione dell'heap (costante quando la dimensione è uguale a 0), e considerando che tale dimensione ad ogni ciclo è 0, 1, 2, ..., $\lfloor \log n \rfloor$, il costo complessivo del `while` risulta essere $O(1) + O(\log 1) + O(\log 2) + \cdots + O(\log \lfloor \log n \rfloor) = O(\log 1 + \log 2 + \cdots + \log \lfloor \log n \rfloor)$. È possibile trasformare la sommatoria di logaritmi in logaritmo di prodotti ottenendo $O(\log(1 \times 2 \times 3 \times \cdots \times \lfloor \log n \rfloor)) = O(\log(\lfloor \log n \rfloor !))$. Il ciclo `for` si comporta in modo simmetrico, eseguendo $\lfloor \log n \rfloor$ cicli, all'interno dei quali si esegue l'operazione `deleteMax`, di complessità logaritmica, sul binary heap che riduce ad ogni ciclo la propria dimensione, iniziando da $\lfloor \log n \rfloor$ e arrivando a 1. Ha quindi il medesimo costo computazionale $O(\log(\lfloor \log n \rfloor !))$. Complessivamente, la complessità risulta quindi essere $T(n) = O(\log(\lfloor \log n \rfloor !))$.

2. Scrivere un algoritmo che prende in input due alberi binari T1 e T2 e restituisce *true* se T1 e T2 sono identici, cioè se hanno la stessa struttura e valori uguali in posizioni corrispondenti (radici uguali, figlio sinistro della radice di T1 uguale al figlio sinistro della radice di T2, figlio destro della radice di T1 uguale al figlio destro della radice di T2, e così via); l'algoritmo restituisce *false* in caso contrario.

Soluzione Per risolvere l'esercizio si può usare una visita ricorsiva "in parallelo" su entrambi gli alberi. Partendo dalle radici di T1 e T2, si verifica se i due nodi contengono lo stesso valore o sono entrambi vuoti. In caso contrario — quindi se uno solo dei due nodi è vuoto o i valori sono diversi — la visita si puó interrompere e il risultato sarà *false*. Se i due nodi hanno valori uguali, gli alberi saranno identici se (i) l'albero radicato nel figlio sinistro di T1 è identico all'albero radicato nel figlio sinistro di T2 e (ii) l'albero radicato nel figlio destro di T1 è identico all'albero radicato nel figlio destro di T2.

```
IDENTICI( nodo T1, nodo T2 ) --> boolean
       if ( T1 == null && T2 == null ) // entrambi i nodi vuoti, quindi identici
              return true;
       else if ( T1 == null || T2 == null ) // uno solo dei due nodi e' vuoto, quindi diversi
              return false;
       else
              return ((T1.val == T2.val) &&
                     IDENTICI(T1.left, T2.left) &&
                     IDENTICI(T1.right, T2.right));
```

La complessità computazione nel caso pessimo è $\Theta(n)$ dove n è il numero di nodi. Questo caso si verifica quando i due alberi sono identici, per cui l'algoritmo esegue (una sola volta) la visita in profondità di entrambi. Il caso ottimo è $O(1)$. Se si assume che l'operatore AND (&&) restituisce *false* appena una condizione è falsa (procedendo da sinistra verso destra) si verifica quando T1 è vuoto. Senza questa assunzione l'algoritmo richiede invece di visitare completamente entrambi gli alberi.

3. Prima di entrare in una sala cinematografica è possibile riempire un contenitore, a forma di bicchiere, con delle caramelle. Il bicchiere può contenere caramelle fino ad un peso complessivo rappresentato da un numero intero K. Le caramelle si prendono da n distributori di caramelle; l'i-esimo distributore, con $1 \leq i \leq n$, contiene caramelle di peso rappresentato da un numero intero $p[i]$. Per comodità, assumiamo che ogni distributore contenga una quantità illimitata di caramelle. Si vuole riempire il bicchiere esattamente per la sua capacità K, massimizzando il peso medio delle caramelle presa. Il peso medio è il peso complessivo delle caramelle prese, diviso il loro numero. Progettare un algoritmo che, data la capacità del bicchiere `K`, ed il vettore `p[1..n]`, restituisce un numero reale indicante il massimo peso medio di caramelle che si possono prendere, assumendo di riempire il bicchiere esattamente per la sua capacità `K`.

Soluzione Al fine di massimizzare il peso medio delle caramelle, l'obiettivo è minimizzare il numero di caramelle usate per raggiungere il peso complessivo K. È possibile calcolare il numero minimo di caramelle per raggiungere il peso complessivo K tramite programmazione dinamica. Usiamo $P(i,j)$ per indicare il numero minimo di caramelle, tra quelle dei distributori con indice minore o uguale a i, per raggiungere il peso complessivo j. Usiamo $P(i,j) = \infty$ per indicare che non è possibile raggiungere il peso complessivo j. Tali problemi $P(i,j)$ possono essere risolti nel seguente modo:

$$P(i,j) = \begin{cases} j/p[1] & \text{se } i = 1 \text{ e } j\%p[1] = 0 \\ \infty & \text{se } i = 1 \text{ e } j\%p[1] \neq 0 \\ P(i-1,j) & \text{se } i > 1 \text{ e } p[i] > j \\ min\{P(i-1,j), 1 + P(i,j-p[i])\} & \textit{altrimenti} \end{cases}$$

Una volta risolto il problema $P(n,K)$, ovvero calcolato il numero minimo di caramelle che permettono di ottenere il peso complessivo K, il peso medio massimo sarà $K/P(n,K)$, che risulterà 0 nel caso in cui non sia possibile riempire il bicchiere esattamente per la sua capacità K.

Il problema può essere risolto dal seguente algoritmo che utilizza la struttura dati ausiliaria `B[i,j]` per memorizzare le soluzioni ai problemi $P(i,j)$.

```
algoritmo Bicchiere(K: Int, p: Int[1..n]) --> Real
  for j = 1 .. K
    if (j % p[1] == 0) then
      B[1,j] = j / p[i]
    else
      B[1,j] = INFINITY
  endfor
  for i = 1 .. n
    for j = 1 .. K
      if (p[i] > j) then
        B[i,j] = B[i-1,j]
      else
        B[i,j] = min{ B[i-1,j], 1+B[i,j-p[i]] }
    endfor
  endfor
  return K / B[n,K]
```

Tutte le operazioni di tale algoritmo hanno costo costante; il primo ciclo `for` ha un contributo sulla complessità pari a $O(K)$, mentre i due `for` annidati hanno un contributo pari a $O(n \cdot K)$. Avremo quindi complessivamente $T(K,n) = O(n \cdot K)$, che diviene $O(n^2)$ nel caso in cui n e K siano del medesimo ordine di grandezza.

4. Progettare un algoritmo che, dato un grafo orientato pesato $G = (V, E, w)$, con tutti i pesi negativi (ovvero, per ogni coppia di vertici adiacenti u_1 e u_2 si ha $w(u_1, u_2) < 0$), e due vertici $s, t \in V$, restituisce, se esiste, il cammino da s a t di peso complessivo massimo.

Soluzione È possibile usare una versione modificata dell'algoritmo di Dijkstra che utilizza una coda con priorità invertita, cioè che restituisce il valore con priorità massima e non il valore con priorità minima. Al termine dell'esecuzione dell'algoritmo di Dijkstra, si stampano i nodi del cammino da s a t nell'albero dei cammini minimi memorizzata nel parent-vector `pred`. Le strutture dati usate sono le solite dell'algoritmo di Dijkstra: il vettore `D` delle distanze e la coda con priorità (massima) `Q`.

Cont.

```
algoritmo CamminoMassimo (Grafo G=(V,E,w), int s, int t)

  // inizializzazione strutture dati
  int n = G.numNodi()
  int pred[1..n], v, u
  double D[1..n]
  for v = 1..n do
    D[v] = -INFINITY
    pred[v] = -1
  endfor
  D[s] = 0
  CodaPriorita<int, double> Q; Q.insert(s, D[s]);

  // esecuzione algoritmi di Dijkstra
  while (not Q.isEmpty()) do
    u = Q.find(); Q.deleteMax()
    for each v adiacente a u do
      if (D[v] == -INFINITY) then
        // prima volta che si incontra v
        D[v] = D[u] + w(u,v)
        Q.insert(v, D[v]);
        pred[v] = u
      elseif (D[u] + w(u,v) > D[v]) then
        // scoperta di un cammino migliore per raggiungere v
        Q.decreaseKey(v, D[v] - D[u] - w(u,v))
        D[v] = D[u] + w(u,v)
        pred[v] = u
      endif
    endfor
  endwhile

  // stampa del cammino da s a t
  printPath(pred,s,t)

// funzione ausiliaria di stampa del cammino massimo
funzione printPath (int pred[], int s, int t)
  if (pred[t] != -1)
    printPath(pred,s,pred[t])
    print t
  endif
```

La complessità dell'algoritmo è la medesima dell'algoritmo di Dijkstra, ovvero $T(n, m) = O(m \log n)$, dove n è il numero di vertici ed m il numero di archi nel grafo.

The End.